Computer Science 523
Advanced Programming
The College of Saint Rose
Summer 2014

Topic Notes: Inheritance

The notes for this topic are very sparse, instead we will follow the examples from Gaddis Chapter 11.

We have seen classes whose class headers include `extends`, `implements`, neither, or both. Our next topic, *inheritance*, will clarify what is happening in those cases and others.

Read Chapter 11 carefully! Below is just a bit of information specific to our upcoming use of interfaces and abstract classes.

Interfaces and Abstract classes

So far, we have seen interfaces and regular classes. There is a level between these called an *abstract class*.

The abstractions provided by interfaces and abstract classes are important for the development of reusable and modular software.

We want to be able to define *what* an abstract data type does without committing to *how* it does it.

The biggest example we've seen so far is a `ArrayList`. As the user of a `ArrayList`, we know we can create them, add, retrieve, remove, and modify elements in them, and query information like their size. All of these are independent of how the `ArrayList` is implemented.

This separation of the public interface from the implementation allows programmers to make use of `ArrayList`s without needing to know how things work on the inside. It also allows the implementers of `ArrayList`s to make internal changes without affecting other code that uses it, so long as the public interface does not change.

Java has language constructs to support the development of abstract data types.

- *Interfaces* describe the public functionality of an abstract data type. This includes:
  - method signatures
  - constants

  An `interface` may `extend` another `interface`.

  We have seen and used `interface`s primarily in the context of listeners for our Swing GUI components.

- *Abstract base classes* describe a partial implementation. An `abstract` class can define method bodies for some of the methods required by `interface`s it `implements`.

  This can be useful for:

  – methods that can be implemented in terms of other methods

  It is possible for a class that extends an `abstract` class to override methods defined in the `abstract` class, in case there is a more efficient way to do some of these things when an actual implementation is developed.

  A frequent use of an `abstract` class is to "factor out" implementation of methods that happen to be the same for multiple implementations of an interface.

- Full implementations (classes that you can instantiate) may `implement interface`s, and/or `extend` exactly one `abstract` or fully implemented class.

We will see some more examples using interfaces and abstract classes as we wrap up this course. You will definitely see more in subsequent courses.